

# TOP-DOWN SKIPLISTS

*Luis Barba\* and Pat Morin<sup>†</sup>*

July 31, 2014

---

**ABSTRACT.** We describe todolists (top-down skiplists), a variant of skiplists (Pugh 1990) that can execute searches using at most  $\log_{2-\varepsilon} n + O(1)$  binary comparisons per search and that have amortized update time  $O(\varepsilon^{-1} \log n)$ . A variant of todolists, called working-todolists, can execute a search for any element  $x$  using  $\log_{2-\varepsilon} w(x) + o(\log w(x))$  binary comparisons and have amortized search time  $O(\varepsilon^{-1} \log w(w))$ . Here,  $w(x)$  is the “working-set number” of  $x$ . No previous data structure is known to achieve a bound better than  $4\log_2 w(x)$  comparisons. We show through experiments that, if implemented carefully, todolists are comparable to other common dictionary implementations in terms of insertion times and outperform them in terms of search times.

---

---

\*School of Computer Science, Carleton University and Département d’Informatique, Université Libre de Bruxelles, [lbarbaf1@ulb.ac.be](mailto:lbarbaf1@ulb.ac.be)

<sup>†</sup>School of Computer Science, Carleton University, [morin@scs.carleton.ca](mailto:morin@scs.carleton.ca)

---

## 1 Introduction

Comparison-based dictionaries supporting the three *basic operations* insert, delete and search represent *the* classic data-structuring problem in computer science. Data structures that support each of these operations in  $O(\log n)$  time have been known since the introduction of AVL trees more than half a century ago [1]. Since then, many competing implementations of dictionaries have been proposed, including red-black trees [15], splay trees [22], general-balanced/scapegoat trees [2, 13], randomized binary search trees [17], energy-balanced trees [14], Cartesian trees/treaps [21, 23], skip lists [19], jump lists [10], and others. Most major programming environments include one or more  $O(\log n)$  time dictionary data structures in their standard library, including Java’s `TreeMap` and `TreeSet`, the C++ STL’s `set`, and Python’s `OrderedDict`.

In short, comparison-based dictionaries are so important that any new ideas or insights about them are worth exploring. In this paper, we introduce the todolist (**top-down skiplist**), a dictionary that is parameterized by a parameter  $\varepsilon \in (0, 1)$ , that can execute searches using at most  $\log_{2-\varepsilon} n + O(1)$  binary comparisons per search, and that has amortized update time  $O(\varepsilon^{-1} \log n)$ . (Note that  $\log_{2-\varepsilon} n \leq (1 + \varepsilon) \log n$  for  $\varepsilon < 1/4$ .)

As a theoretical result todolists are nothing to get excited about; there already exist comparison-based dictionaries with  $O(\log n)$  time for all operations that perform at most  $\lceil \log n \rceil + 1$  comparisons per operation [3, 12]. (Here, and throughout,  $\log n = \log_2 n$  denotes the binary logarithm of  $n$ . However, todolists are based on a new idea—top-down partial rebuilding of skiplists—and our experimental results show that a careful implementation of todolists can execute searches faster than existing popular data structures.

In particular, todolists outperform (again, in terms of searches) Guibas and Sedgewick’s red-black trees [15] which are easily the most common implementation of comparison-based dictionaries found in programming libraries. This is no small feat since, in the setting we studied, the average depth of a node in a red-black tree seems to be  $\log n - O(1)$  [20].

As a more substantial theoretical contribution, we show that a variant of todolists, called *working-todolists*, is able to search for an element  $x$  using  $\log_{2-\varepsilon} w(x) + o(\log w(x))$  comparisons in  $O(\varepsilon^{-1} \log w(x))$  amortized time. Here,  $w(x)$ —the *working set number* of  $x$ —is loosely defined as the number of distinct items accessed

---

since the last time  $x$  was accessed (see Section 3 for a precise definition.) Previous data structures with some variant of this *working-set property* include splay trees [22], Iacono’s working-set structure [16, 4], deterministic self-adjusting skiplists [9], layered working-set trees [8], and skip-splay trees [11]. However, even the most efficient of these can only be shown to perform at most  $4\log w(x)$  comparisons during a search for  $x$ .

## 2 TodoLists

A *todolist* for the values  $x_1 < x_2 < \dots < x_n$  consists of a nested sequence of  $h + 1$  sorted singly-linked lists,  $L_0, \dots, L_h$ , having the following properties:<sup>1</sup>

1.  $|L_0| \leq 1$ .
2.  $L_i \subseteq L_{i+1}$  for each  $i \in \{0, \dots, h-1\}$ .
3. For each  $i \in \{1, \dots, h\}$  and each pair  $x, y$  of consecutive elements in  $L_i$ , at least one of  $x$  or  $y$  is in  $L_{i-1}$ .
4.  $L_h$  contains  $x_1, \dots, x_n$ .

The value of  $h$  is at least  $\lceil \log_{2-\epsilon} n \rceil$  and at most  $\lceil \log_{2-\epsilon} n \rceil + 1$ . The *height* of a value,  $x$ , in a todolist is the number of lists in which  $x$  appears.

We will assume that the head of each list  $L_i$  is a *sentinel* node that does not contain any data. (See Figure 1.) We will also assume that, given a pointer to the node containing  $x_j$  in  $L_i$ , it is possible to find, in constant time, the occurrence of  $x_j$  in  $L_{i+1}$ . This can be achieved by maintaining an extra pointer or by maintaining all occurrences of  $x_j$  in an array. (See Section 4 for a detailed description.)

### 2.1 Searching

Searching for a value,  $x$ , in a todolist is simple. In particular, we can find the node,  $u$ , in  $L_h$  that contains the largest value that is less than  $x$ . If  $L_h$  has no value less than  $x$  then the search finds the sentinel in  $L_h$ . We call the node  $u$  the *predecessor* of  $x$  in  $L_h$ .

Starting at the sentinel in  $L_0$ , one comparison (with the at most one element of  $L_0$ ) is sufficient to determine the predecessor,  $u_0$  of  $x$  in  $L_0$ . (This follows from

---

<sup>1</sup>Here and throughout, we use set notations like  $|\cdot|$ , and  $\subseteq$  on the lists  $L_0, \dots, L_h$ , with the obvious interpretations.

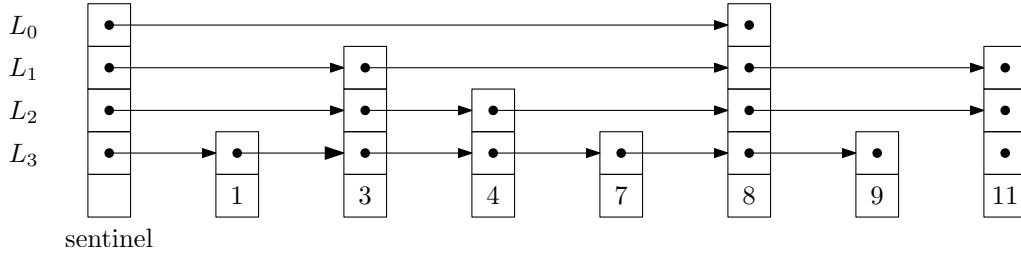


Figure 1: An example of a todolist containing 1, 3, 4, 7, 8, 9, 11.

Property 1.) Moving down to the occurrence of  $u_0$  in  $L_1$ , one additional comparison is sufficient to determine the predecessor,  $u_1$  of  $x$  in  $L_1$ . (This follows from Property 3.) In general, once we know the predecessor of  $x$  in  $L_i$  we can determine the predecessor of  $x$  in  $L_{i+1}$  using one additional comparison. Thus, the total number of comparisons needed to find the predecessor of  $x$  in  $L_h$  is only  $h + 1$ .

**FINDPREDECESSOR( $x$ )**

```

 $u_0 \leftarrow \text{sentinel}_0$ 
for  $i = 0, \dots, h$  do
    if  $\text{next}(u_i) \neq \text{nil}$  and  $\text{key}(\text{next}(u_i)) < x$  then
         $u_i \leftarrow \text{next}(u_i)$ 
         $u_{i+1} \leftarrow \text{down}(u_i)$ 
return  $u_h$ 

```

## 2.2 Adding

Adding a new element,  $x$ , to a todolist is done by searching for it using the algorithm outlined above and then splicing  $x$  into each of the lists  $L_0, \dots, L_h$ . This splicing is easily done in constant time per list, since the new nodes containing  $x$  appear after the nodes  $u_0, \dots, u_h$ . At this point, all of the Properties 2–4 are satisfied, but Property 1 may be violated since there may be two values in  $L_0$ .

If there are two values in  $L_0$ , then we restore Property 1 with the following *partial rebuilding* operation: We find the smallest index  $i$  such that  $|L_i| \leq (2 - \varepsilon)^i$ ; such an index always exists since  $n = |L_h| \leq (2 - \varepsilon)^h$ . We then rebuild the lists  $L_0, \dots, L_{i-1}$  in a bottom up fashion;  $L_{i-1}$  gets every second element from  $L_i$  (starting with the second),  $L_{i-2}$  gets every second element from  $L_{i-1}$ , and so on down to  $L_0$ .

Since we take every other element from  $L_i$  starting with the second element,

---

after rebuilding we obtain:

$$|L_{i-1}| = \lfloor |L_i|/2 \rfloor \leq |L_i|/2$$

and, repeating this reasoning for  $L_{i-2}, L_{i-3}, \dots, L_0$ , we see that, after rebuilding,

$$|L_0| \leq |L_i|/2^i \leq (2 - \varepsilon)^i/2^i < 1 \ .$$

Thus, after this rebuilding,  $|L_0| = 0$ , Property 1 is restored and the rebuilding, by construction, produces lists satisfying Properties 2–4.

To study the amortized cost of adding an element, we can use the potential method with the potential function

$$\Phi(L_0, \dots, L_h) = C \sum_{i=0}^h |L_i| \ .$$

Adding  $x$  to each of  $L_0, \dots, L_h$  increases this potential by  $C(h+1) = O(C \log n)$ . Rebuilding, if it occurs, takes  $O(|L_i|) = O((2 - \varepsilon)^i)$  time, but causes a change in potential of at least

$$\begin{aligned} \Delta\Phi &= C \sum_{j=0}^i (|L_j| - (2 - \varepsilon)^j) \\ &= C \sum_{j=0}^i (|L_i|/2^{i-j} - (2 - \varepsilon)^j) \\ &\leq C \sum_{j=0}^{i-1} ((2 - \varepsilon)^i/2^{i-j} - (2 - \varepsilon)^j) \\ &\leq C \left( (2 - \varepsilon)^i - \sum_{j=0}^{i-1} (2 - \varepsilon)^j \right) \\ &= C \left( (2 - \varepsilon)^i - \frac{(2 - \varepsilon)^i - (2 - \varepsilon)}{1 - \varepsilon} \right) \\ &< C \left( (2 - \varepsilon)^i - (1 + \varepsilon) \left( (2 - \varepsilon)^i - (2 - \varepsilon) \right) \right) \quad (\text{since } 1/(1 - \varepsilon) > 1 + \varepsilon) \\ &= -C\varepsilon(2 - \varepsilon)^i + O(C) \end{aligned}$$

Therefore, by setting  $C = c/\varepsilon$  for a sufficiently large constant,  $c$ , the decrease in potential is greater than the cost of rebuilding. We conclude that the amortized cost of adding an element  $x$  is  $O(C \log n) = O(\varepsilon^{-1} \log n)$ .

---

---

### 2.3 Deleting

Since we already have an efficient method of partial rebuilding, we can use it for deletion as well. To delete an element  $x$ , we delete it in the obvious way, by searching for it and then splicing it out of the lists  $L_i, \dots, L_h$  in which it appears. At this point, Properties 1, 2, and 4 hold, but Property 3 may be violated in any subset of the lists  $L_i, \dots, L_h$ . Luckily, all of these violations can be fixed by taking  $x$ 's successor in  $L_h$  and splicing it into each of  $L_0, \dots, L_{h-1}$ .<sup>2</sup> Thus, the second part of the deletion operation is like the second part of the insertion operation. Like the insertion operation, this may violate Property 1 and trigger a partial rebuilding operation. The same analysis used to study insertion shows that deletion has the same amortized running time of  $O(\varepsilon^{-1} \log n)$ .

### 2.4 Tidying Up

Aside from the partial rebuilding caused by insertions and deletions, there are also some *global rebuilding* operations that are sometimes triggered:

1. If an insertion causes  $n$  to exceed  $\lceil (2 - \varepsilon)^h \rceil$ , then we increment the value of  $h$  to  $h' = h + 1$  and rebuild  $L_0, \dots, L_{h'}$  from scratch, starting by moving  $L_h$  into  $L_{h'}$  and then performing a partial rebuilding operation on  $L_0, \dots, L_{h'-1}$ .
2. If an insertion or deletion causes  $\sum_{i=1}^n |L_i|$  to exceed  $cn$  for some threshold constant  $c > 2$ , then we perform a partial rebuilding to rebuild  $L_0, \dots, L_{h-1}$ .
3. If a deletion causes  $n$  to be less than  $\lceil (2 - \varepsilon)^{h-2} \rceil$  then we decrement the value of  $h$  to be  $h' = h - 1$ , move  $L_h$  to  $L_{h'}$  and then perform a partial rebuilding operation on  $L_0, \dots, L_{h'-1}$ .

A standard amortization argument shows that the first and third type of global rebuilding contribute only  $O(1)$  to the amortized cost of each insertion and deletion, respectively. The same potential function argument used to study insertion and deletion works to show that the second type of global rebuilding contributes only  $O(\log n)$  to the amortized cost of each insertion or deletion (note that this second type of global rebuilding is only required to ensure that the size of the data structure remains in  $O(n)$ ).

This completes the proof of our first theorem:

---

<sup>2</sup>If  $x$  has no successor in  $L_h$ —because it is the largest value in the to-dolist—then deleting  $x$  will not introduce any violations of Property 3.

---

**Theorem 1.** *For any  $\varepsilon > 0$ , a todolist supports the operations of inserting, deleting, and searching using at most  $\log_{2-\varepsilon} n + O(1)$  comparisons per operation. Starting with an empty todolist and performing any sequence of  $N$  add and remove operations takes  $O(\varepsilon^{-1} N \log N)$  time.*

### 3 Working-TodoLists

Next, we present a new theoretical result that is achieved using a variant of the todolist that we call a working-todolist. First, though, we need some definitions. Let  $a_1, \dots, a_m$  be a sequence whose elements come from the set  $\{1, \dots, n\}$ . We call such a sequence an *access sequence*. For any  $x \in \{1, \dots, n\}$ , the *last-occurrence*,  $\ell_t(x)$ , of  $x$  at time  $t$  is defined as

$$\ell_t(x) = \max\{j \in 1, \dots, t-1 : a_j = x\} .$$

Note that  $\ell_t(x)$  is undefined if  $x$  does not appear in  $a_1, \dots, a_{t-1}$ . The *working-set number*,  $w_t(x)$ , of  $x$  at time  $t$  is

$$w_t(x) = \begin{cases} n & \text{if } \ell_t(x) \text{ is undefined} \\ |\{a_{\ell_t(x)}, \dots, a_{t-1}\}| & \text{otherwise.} \end{cases}$$

In words, if we think of  $t$  as the current time, then  $w_t(x)$  is the number of distinct values in the access sequence since the most recent access to  $x$ .

In this section, we describe the working-todolist data structure, which stores  $\{1, \dots, n\}$  and, for any access sequence  $a_1, \dots, a_m$ , can execute searches for  $a_1, \dots, a_m$  so that the search for  $a_t$  performs at most  $(1 + o(1)) \log_{2-\varepsilon} w_t(a_t)$  comparisons and takes  $O(\varepsilon^{-1} \log w_t(a_t))$  amortized time.

From this point onward we will drop the time subscript,  $t$ , on  $w_t$  and assume that  $w(x)$  refers to the working set number of  $x$  at the current point in time (given the sequence of searches performed so far). The working-todolist is a special kind of todolist that weakens Property 1 and adds an additional Property 5:

1.  $|L_0| \leq \varepsilon^{-1} + 1$ .
5. For each  $i \in \{0, \dots, h\}$ ,  $L_i$  contains all values  $x$  such that  $w(x) \leq (2 - \varepsilon)^i$ .

For keeping track of working set numbers, a working-todolist also stores a doubly-linked list,  $Q$ , that contains the values  $\{1, \dots, n\}$  ordered by their current working set numbers. The node that contains  $x$  in this list is cross-linked (or associated in some other way) with the appearances of  $x$  in  $L_0, \dots, L_h$ .

---

### 3.1 Searching

Searching in a working-todolist is similar to a searching in a todolist. The main difference is that Property 5 guarantees that the working-todolist will reach a list,  $L_i$ , that contains  $x$  for some  $i \leq \log_{2-\varepsilon} w(x)$ . If ternary comparisons are available, then this is detected at the first such index  $i$ . If only binary comparisons are available, then the search algorithm is modified slightly so that, at each list  $L_i$  where  $i$  is a perfect square, an extra comparison is done to test if the successor of  $x$  in  $L_i$  contains the value  $x$ . This modification ensures that, if  $x$  appears first in  $L_i$ , then it is found by the time we reach the list  $L_{i'}$  for

$$i' = i + \lceil 2\sqrt{i} \rceil + 1 = \log_{2-\varepsilon} w(x) + O(\sqrt{\log w(x)}) .$$

Once we find  $x$  in some list  $L_{i'}$ , we move it to the front of  $Q$ ; this takes only constant time since the node containing  $x$  in  $L_{i'}$  is cross-linked with  $x$ 's occurrence in  $Q$ . Next, we insert  $x$  into  $L_0, \dots, L_{i'-1}$ . As with insertion in a todolist, this takes only constant time for each list  $L_j$ , since we have already seen the predecessor of  $x$  in  $L_j$  while searching for  $x$ . At this point, Properties 2–5 are ensured and the ordering of  $Q$  is correct.

All that remains is to restore Property 1, which is now violated since  $L_0$  contains  $x$ , for which  $w(x) = 1$ , and the value  $y$  such that  $w(y) = 2$ . Again, this is corrected using partial rebuilding, but the rebuilding is somewhat more delicate. We find the first index  $i$  such that  $|L_i| \leq (2-\varepsilon/2)^i$ . Next, we traverse the first  $(2-\varepsilon)^{i-1}$  nodes of  $Q$  and label them with their position in  $Q$ . Since  $Q$  is ordered by working-set number, this means that the label at a node of  $Q$  that contains the value  $z$  is at most  $w(z)$ .

At this point, we are ready to rebuild the lists  $L_0, \dots, L_{i-1}$ . To build  $L_{j-1}$  we walk through  $L_j$  and take any value whose label (in  $Q$ ) is defined and is at most  $(2-\varepsilon)^j$  as well as every “second value” as needed to ensure that Property 3 holds. Finally, once all the lists  $L_0, \dots, L_j$  are rebuilt, we walk through the first  $(2-\varepsilon)^{i-1}$  nodes of  $Q$  and remove their labels so that these labels are not incorrectly used during subsequent partial rebuilding operations.

### 3.2 Analysis

We have already argued that we find a node containing  $x$  in some list  $L_i$  with  $i \in \log_{2-\varepsilon} w(x) + O(\sqrt{\log w(x)})$  and that this takes  $O(\log w(x))$  time. The number of



---

comparisons needed to reach this stage is

$$\log_{2-\varepsilon} w(x) + O\left(\varepsilon^{-1} + \sqrt{\log w(x)}\right) .$$

The  $O(\varepsilon^{-1})$  term is the cost of searching in  $L_0$  and the  $O(\sqrt{\log w(x)})$  term accounts for one comparison at each of the lists  $L_{\lceil \log_{2-\varepsilon} w(x) \rceil}, \dots, L_i$  as well as the extra comparison performed in each of the lists  $L_j$  where  $j \in \{0, \dots, i\}$  is a perfect square.

After finding  $x$  in  $L_i$ , the algorithm then updates  $L_0, \dots, L_{i-1}$  in such a way that Properties 2–5 are maintained. All that remains is to show that Property 1 is restored by the partial rebuilding operation and to study the amortized cost of this partial rebuilding. We accomplish both these goals by studying the sizes of the lists  $L_0, \dots, L_i$  after rebuilding.

Let  $n_i = |L_i|$  and recall that  $n_i \leq (2 - \varepsilon/2)^i$ . Then, the number of elements that make it from  $L_i$  into  $L_{i-1}$  is

$$|L_{i-1}| \leq (2 - \varepsilon)^{i-1} + n_i/2 ,$$

and the number of elements that make it into  $L_{i-2}$  is

$$\begin{aligned} |L_{i-2}| &\leq (2 - \varepsilon)^{i-2} + |L_{i-1}|/2 \\ &\leq (2 - \varepsilon)^{i-2} + (2 - \varepsilon)^{i-1}/2 + n_i/4 . \end{aligned}$$

More generally, the number of elements that make it into  $L_j$  for any  $j \in \{0, \dots, i-1\}$  is at most

$$\begin{aligned} |L_j| &\leq (2 - \varepsilon)^j \cdot \sum_{k=0}^{i-j-1} \left(\frac{2 - \varepsilon}{2}\right)^k + n_i/2^{i-j} \\ &\leq (2 - \varepsilon)^j/\varepsilon + n_i/2^{i-j} . \end{aligned}$$

In particular

$$|L_0| \leq \varepsilon^{-1} + n_i/2^i \leq \varepsilon^{-1} + 1 .$$

Therefore, Property 1 is satisfied.

To study the amortized cost of searching for  $x$ , we use the same potential

---

function argument as in Section 2. The change in the sizes of the lists is then

$$\begin{aligned}
\Delta\Phi/C &\leq \sum_{j=0}^{i-1} \left( (2-\varepsilon)^j/\varepsilon + n_i/2^{i-j} - (2-\varepsilon/2)^j \right) \\
&\leq O((2-\varepsilon)^i/\varepsilon) + n_i - \sum_{j=0}^{i-1} (2-\varepsilon/2)^j \\
&= O((2-\varepsilon)^i/\varepsilon) + n_i - \frac{(2-\varepsilon/2)^i}{1-\varepsilon/2} + O(1) \\
&\leq O((2-\varepsilon)^i/\varepsilon) + n_i - (1+\varepsilon/2)((2-\varepsilon/2)^i) \\
&\leq O((2-\varepsilon)^i/\varepsilon) + n_i - (1+\varepsilon/2)n_i \\
&\leq O((2-\varepsilon)^i/\varepsilon) - (\varepsilon/2)n_i \\
&= -\Omega(\varepsilon n_i) \qquad \qquad \qquad (\text{since } n_i \geq n_{i-1} \geq (2-\varepsilon/2)^{i-1})
\end{aligned}$$

Since the cost of rebuilding  $L_0, \dots, L_{i-1}$  is  $O(n_i)$ , this implies that the amortized cost of accessing  $x$  is  $O(\varepsilon^{-1} \log w(x))$ .

## 4 Implementation Issues

As a first attempt, one might try to implement a todolist exactly as described in Section 2, with each list  $L_i$  being a separate singly linked list in which each node has a down pointer to the corresponding node in  $L_{i+1}$ . However, past experience with skiplists suggests (and preliminary experiments confirms) that this is neither space-efficient nor fast (see the class `LinkedTodoList` in the source code). Instead, we use an implementation idea that appears in Pugh's original paper on skiplists [19].

### 4.1 Nodes as Arrays

A better implementation uses one structure for each data item,  $x$ , and this structure includes an array of pointers. If  $x$  appears in  $L_i, \dots, L_h$ , then this array has length  $h-i+1$  so that it can store the next pointers for the occurrence of  $x$  in each of these lists.

One complication occurs in implementing arrays of next pointers in a todolist that is not present in a skiplist. During a partial rebuilding operation, the heights of elements in a todolist change, which means that their arrays need to be reallocated. The cost of reallocating and initializing an array is proportional to the length of the array. However, the amortization argument used in Section 2.2,

---

requires that the cost of increasing the height of an element when rebuilding level  $i$  is proportional to the increase in height; promoting an element from level  $i$  to level  $i + c$  should take  $O(c)$  time, not  $O(h - i + c)$  time.

The work-around for this problem is to use a standard doubling trick used to implement dynamic arrays (c.f., Morin [18, Section 2.1.2]). When a new array for a node is allocated to hold  $r$  values, its size is set to  $r' = 2^{\lceil \log r \rceil}$ . Later, if the height of the node increases, to  $r + c$  during a partial rebuilding operation, the array only needs to be reallocated if  $r + c > r'$ . Using this trick, the amortized cost of increasing the height of the node is  $O(c)$ . This trick does not increase the number of pointers in the structure by more than factor of 2.

Our initial implementation did exactly this, and performed well-enough to execute searches faster than standard skiplists but was still bested by most forms of binary search trees. This was despite the fact that the code for searching was dead-simple, and by decreasing  $\varepsilon$  we could reduce the height,  $h$ , (and hence the number of comparisons) to less than was being performed by these search trees.

## 4.2 The Problem With Skiplists

After some reflection, the reason for the disappointing performance of searches in todolists (and to a greater extent, in skiplists) became apparent. It is due to the fact that accessing a node by following a pointer that causes a CPU cache miss is more expensive than performing a comparison.

The search path in a todolist has length equal to the number of comparisons performed. However, the set of nodes in the todolist that are dereferenced during a search includes nodes not on the search path. Indeed, when the outcome of a comparison of the form  $\text{key}(\text{next}(u)) < x$  is false, the search path proceeds to  $\text{down}(u)$  and the node  $\text{next}(u)$  does not appear on the search path.

## 4.3 The Solution

Luckily, there is a fairly easy remedy, though it does use more space. We implement the todolist so that each node  $u$  in a list  $L_i$  stores an additional key,  $\text{keynext}(u)$ , that is the key associated with the node  $\text{next}(u)$ . This means that determining the next node to visit after  $u$  can be done using the key,  $\text{keynext}(u)$ , stored at node  $u$  rather than having to dereference  $\text{next}(u)$ . The resulting structure is illustrated in Figure 2.

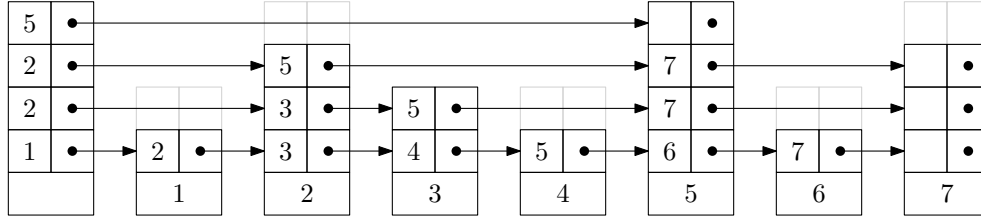


Figure 2: The memory layout of an efficient todolist implementation.

With this modification, todolists achieve faster search times—even with fairly large values of  $\varepsilon$ —than binary search tree structures. Indeed, retrofitting this idea into the skiplist implementation improves its performance considerably so that it outperforms some (but not all) of the tree-based structures.

#### 4.4 Experiments

To test the performance of todolists, we implemented them and tested them against other comparison-based dictionaries that are popular, either in practice (red-black trees) and/or in textbooks (scapegoat trees, treaps, and skiplists). The implementation of all data structures was done in C++ and all the code was written by the second author.<sup>3</sup> To ensure that this code is comparable to so-called industrial strength C++ code, the tests also include the C++ Standard Template Library set class that comes as part of `libstdc++`. This set class is implemented as a red-black tree and performed indistinguishably from our red-black tree implementation.

The code used to generate all the test data in this section is available for download at [github](https://github.com/patmorin/todolist).<sup>4</sup>

The experimental data in this section was obtained from the program `main.cpp` that can be found in the accompanying source code. This program was compiled using the command line: `g++ -std=c++11 -Wall -O4 -o main main.cpp`. The compiler was the `gcc` compiler, version 4.8.2 that ships with the Ubuntu 14.04 Linux distribution. Tests were run on a desktop computer having 16GB DDR3 1600MHz memory and a Intel Core i5-4670K processor with 6MB L3 cache and running at 3.4GHz.

<sup>3</sup>The implementations of all but todolists were adapted from the second author’s textbook [18].

<sup>4</sup>The source code is available at <https://github.com/patmorin/todolist>. The final version of this paper will provide a digital object identifier (DOI) that provides a link to the permanent fixed version of the source code that was used to generate all data in the final paper.

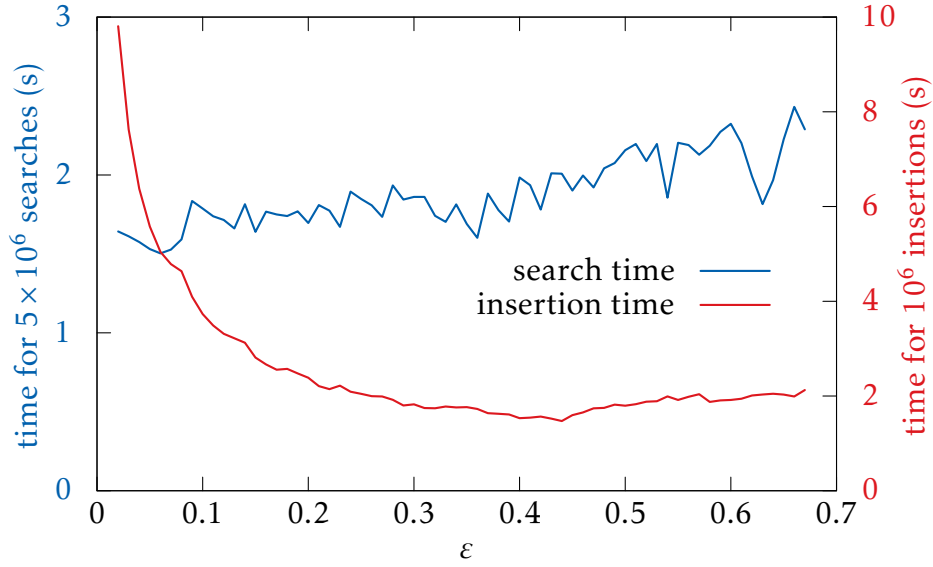


Figure 3: The trade-off between search time and insertion time as a function of  $\varepsilon$ .

#### 4.4.1 Varying $\varepsilon$

Figure 3 shows the results of varying the value of  $\varepsilon$  from 0.02 to 0.68 in increments of 0.01. In this figure,  $n = 10^6$  random integers in the set  $\{0, 5, \dots, 5(n-1)\}$  were chosen (with replacement) and inserted into a todolist. Since dictionaries discard duplicates, the resulting todolist contained 906,086 values. This todolist was then searched  $m = 5n$  times with random integers chosen, with replacement, from  $\{-2, \dots, 5n+3\}$ . This figure illustrates that todolists do behave roughly as Theorem 1 predicts. Insertion time increases roughly proportionally to  $1/\varepsilon$  and search times seem to be of the form  $c(d + \varepsilon)$  for some constant  $c$  and  $d$  (though there is certainly lots of noise in the search times).

In terms of implementation guidance, this figure suggests that values of  $\varepsilon$  below 0.1 are hard to justify. The improvement in search time does not offset the increase in insertion time. At the same time, values of  $\varepsilon$  greater than 0.35 do not seem to be of much use either since they increase the search time and don't decrease the insertion time significantly. At some point beyond this—at around  $\varepsilon = 0.45$ —increasing  $\varepsilon$  increases both the insertion time and the search time (since every insertion starts with a search).

---

#### 4.4.2 The Race

Next, we tested the performance of todolists against a number of other common dictionary data structures, including skiplists, red-black trees, scapegoat trees, and treaps. As a baseline, we also measured the search performance of two static data structures: sorted arrays, and perfectly balanced binary search trees.

In these tests, the value of  $n$  varied from 25,000 to  $2 \times 10^6$  in increments of 25,000. Each individual test followed the same pattern as in the previous section and consisted of  $n$  insertions followed by  $5n$  searches.

**Searches: Todolists win.** The timing results for the searches are shown in Figure 4. In terms of search times, todolists—with  $\varepsilon = 0.2$  and  $\varepsilon = 0.35$ —are the winners among all the dynamic data structures, and even match the performance of statically-built perfectly-balanced binary search trees. The next fastest dynamic data structures are red-black trees which, for larger  $n$  have a search time of roughly 1.4 times that of perfectly-balanced binary search trees.

Surprisingly, todolists beat red-black trees because of their memory layout, not because they reduce the number of comparisons. In these experiments, the average number of comparisons done by red-black trees during a search was measured to be  $\alpha \log n$  for  $\alpha \in [1.02, 1.03]$ .<sup>5</sup> This is substantially less than the number of comparisons done by todolists, which is about  $1.2 \log n$  (for  $\varepsilon = 0.2$ ) and  $1.35 \log n$  (for  $\varepsilon = 0.35$ ). The optimal binary search trees also have a similarly efficient memory layout because the algorithm that constructs them allocates nodes in the order they are encountered in a pre-order traversal. A consequence of this is that the left child of any node,  $u$ , is typically placed in a memory location adjacent to  $u$ . Thus, during a random search, roughly half the steps proceed to an adjacent memory location.

**Insertions: Todolists lose.** The timing results for the insertions are shown in Figure 5. This is where the other shoe drops. Even with  $\varepsilon = 0.35$ , insertions take about three to four times as long in a todolist as in a red-black tree. Profiling the code shows that approximately 65% of this time is spent doing partial rebuilding and another 6% is due to global rebuilding.

---

<sup>5</sup>This incredibly good performance of red-black trees created by inserting random data has been observed already by Sedgwick [20], who conjectures that the average depth of a node in a such a red-black tree is  $\log n - 1/2$ .

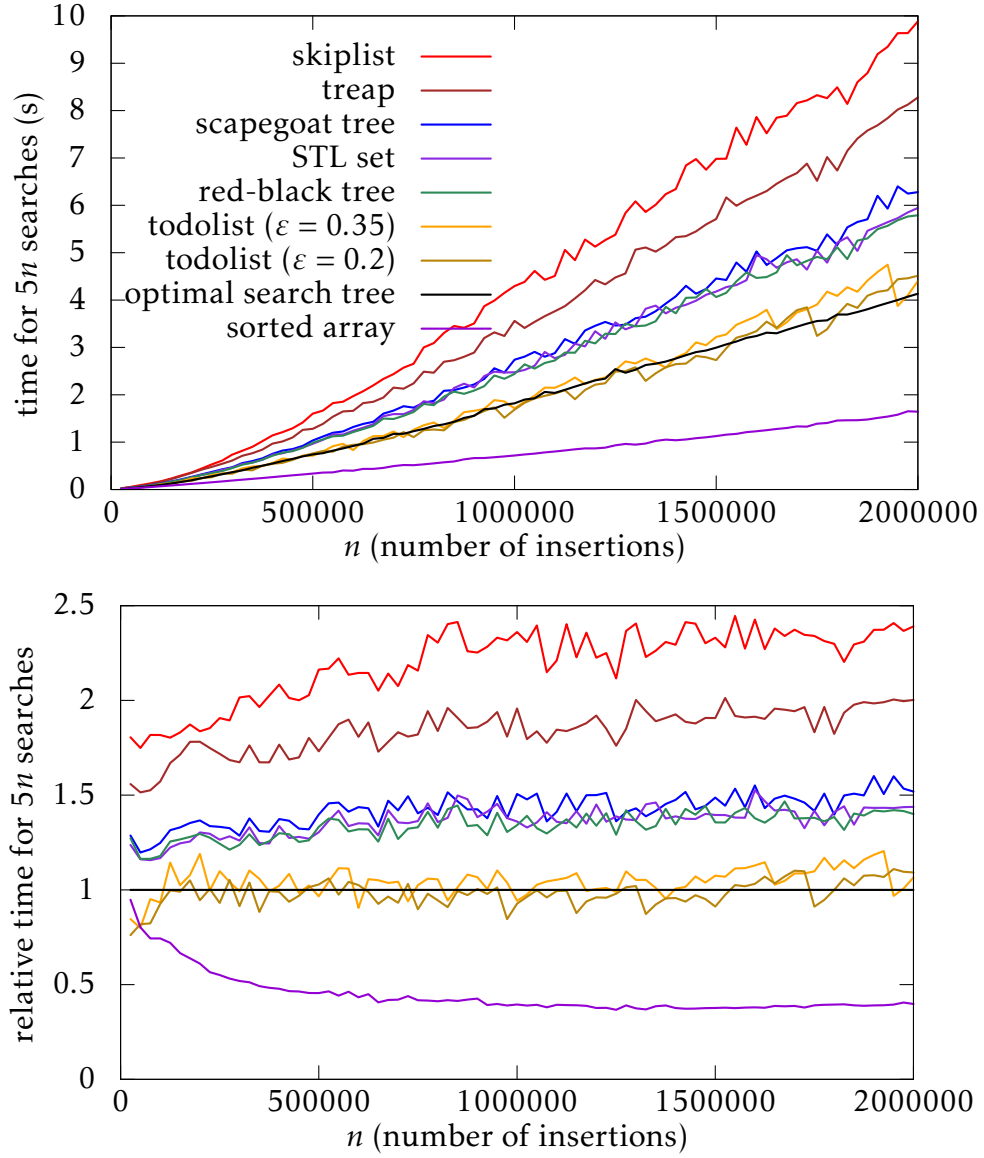


Figure 4: Search time comparison between different data structures. The top graph shows absolute times, in seconds. The bottom graph shows relative times, normalized by the time taken in the optimal search tree.

---

One perhaps surprising result is that scapegoat trees, which are also based on partial rebuilding, outperform todolists in terms of insertions. This is because scapegoat trees are opportunistic, and only perform partial rebuilding when needed to maintain a small height. Randomly built binary search trees have logarithmic depth, so scapegoat trees do very little rebuilding in our tests. In a similar test that inserts elements in increasing order, scapegoat tree insertions took approximately 50% longer than todolist insertions.

## 5 Conclusion

If searches are much more frequent than updates, then todolists may be the right data structure to use. When implemented properly, their search times are difficult to beat. They perform  $\log_{2-\epsilon} n$  comparisons and roughly half these lead to an adjacent array location. Thus, a search in a todolist should incur only about  $\frac{1}{2} \log_{2-\epsilon} n$  cache misses on average.  $B$ -trees [5] and their cache-oblivious counterparts [6, 7] can reduce this to  $O(\log_B n)$ , where  $B$  is the size of a cache line, but they have considerably higher implementation complexity and running-time constants.

On the other hand, todolists leave a lot to be desired in terms of insertion and deletion time. Like other structures that use partial rebuilding, the restructuring done during an insertion takes time  $\Omega(\log n)$ , so is non-negligible. The implementation of the insertion algorithm used in our experiments is fairly naïve and could probably be improved, but it seems unlikely that its performance will ever match that of, for example, red-black trees.

## Acknowledgement

The authors are grateful to Rolf Fagerberg for helpful discussions and suggestions.

## References

- [1] Adelson-Velskii and Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962. English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259–1263, 1962.
- [2] Arne Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999.
- [3] Arne Andersson and Tony W. Lai. Fast updating of well-balanced trees. In



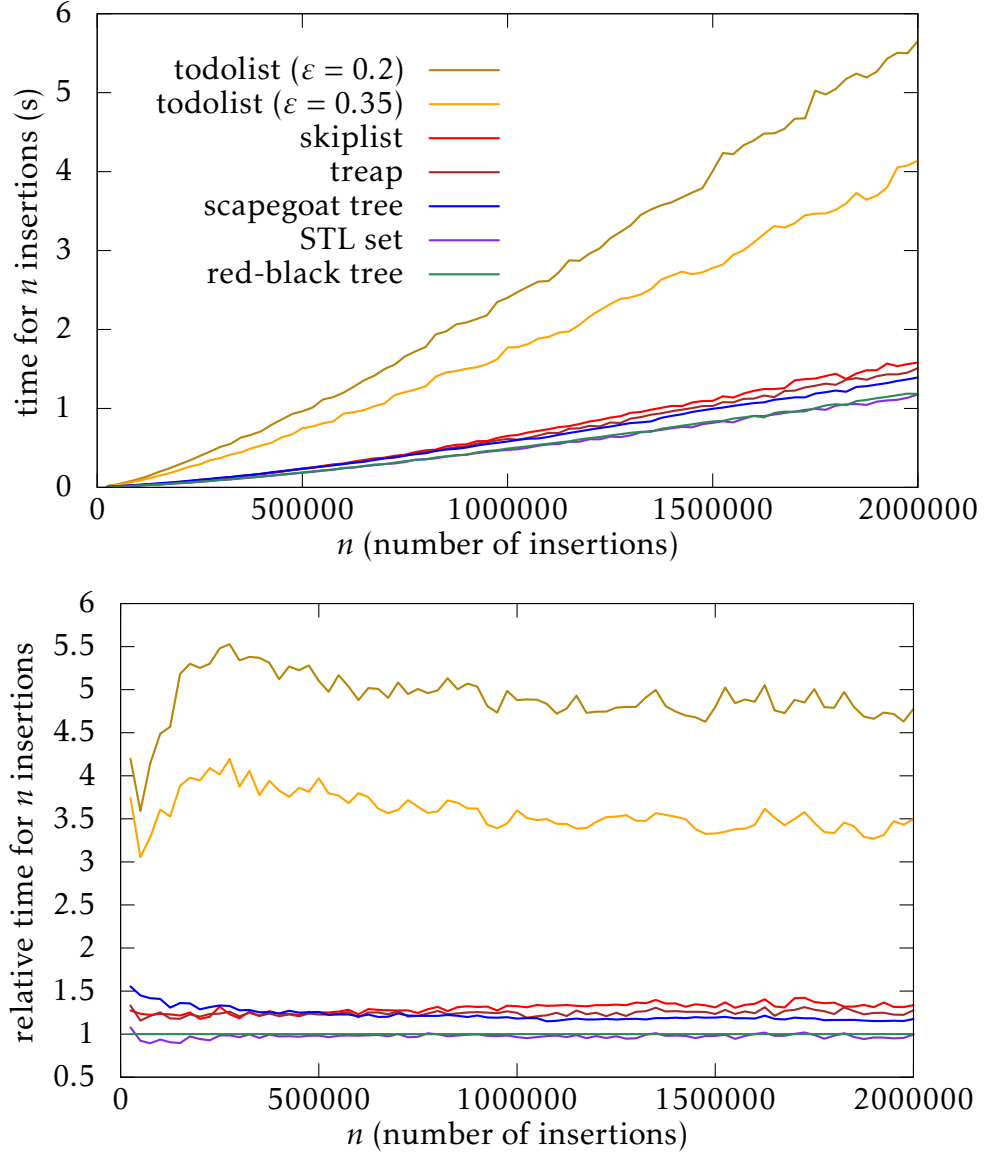


Figure 5: Insertion time comparison between different data structures. The top graph shows absolute times, in seconds. The bottom graph shows relative times, normalized by the running time of the red-black tree.

- 
- John R. Gilbert and Rolf G. Karlsson, editors, *SWAT*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 1990.
- [4] Mihai Badoiu, Richard Cole, Erik D. Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theor. Comput. Sci.*, 382(2):86–96, 2007.
  - [5] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.
  - [6] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2):341–358, 2005.
  - [7] Michael A. Bender, Ziyang Duan, John Iacono, and Jing Wu. A locality-preserving cache-oblivious dynamic dictionary. *J. Algorithms*, 53(2):115–136, 2004.
  - [8] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered working-set trees. *Algorithmica*, 63(1-2):476–489, 2012.
  - [9] Prosenjit Bose, Karim Douïeb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In Shang-Hua Teng, editor, *SODA*, pages 1106–1114. SIAM, 2008.
  - [10] Hervé Brönnimann, Frédéric Cazals, and Marianne Durand. Randomized jumplists: A jump-and-walk dictionary data structure. In Helmut Alt and Michel Habib, editors, *STACS*, volume 2607 of *Lecture Notes in Computer Science*, pages 283–294. Springer, 2003.
  - [11] Jonathan Derryberry and Daniel Dominic Sleator. Skip-splay: Toward achieving the unified bound in the bst model. In Frank K. H. A. Dehne, Marina L. Gavrilova, Jörg-Rüdiger Sack, and Csaba D. Tóth, editors, *WADS*, volume 5664 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2009.
  - [12] Rolf Fagerberg. Binary search trees: How low can you go? In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *Lecture Notes in Computer Science*, pages 428–439. Springer, 1996.
  - [13] Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *SODA*, pages 165–174. ACM/SIAM, 1993.
-

- 
- [14] Michael T. Goodrich. Competitive tree-structured dictionaries. In David B. Shmoys, editor, *SODA*, pages 494–495. ACM/SIAM, 2000.
  - [15] Leonidas J. Guibas and Robert Sedgwick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE Computer Society, 1978.
  - [16] John Iacono. Alternatives to splay trees with  $o(\log n)$  worst-case access times. In S. Rao Kosaraju, editor, *SODA*, pages 516–522. ACM/SIAM, 2001.
  - [17] Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998.
  - [18] Pat Morin. *Open Data Structures: An Introduction*. Athabasca University Press, Edmonton, 2013. C++ code available at <http://opendatastructures.org/>.
  - [19] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
  - [20] Robert Sedgwick. Left-leaning red-black trees. Manuscript.
  - [21] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.
  - [22] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
  - [23] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.